LINUX TOOLCHAIN OVERVIEW

DEBUGGING (GDB), TRACING (LTTNG) FEATURES TOOL WORK GROUP

DOMINIQUE <DOT> TOUPIN <AT> ERICSSON <DOT> COM

EMBEDDED LINUX CONFERENCE, APRIL 2010

ABSTRACT

With new advanced debug and trace features, developer will find troubleshooting with printf very archaic. Developers can now use debug features like reversible debug, record and replay, multi-process, multi-architecture, multi-operating system, non-stop, global breakpoint, core-awareness, even dynamic tracepoint on a live system. For troubleshooting a live system without causing overhead, static tracepoints now offer a rich set of features. In the past year, those features have been introduced in open source tools. This presentation will describe those features with GDB, LTTng, the Eclipse Debugger Services Framework, the Eclipse Tracing Framework and will give an overview of the whole Linux toolchain integration with GNU and Eclipse tools.

ABOUT ME

- Developer Tool Manager at Ericsson, helping Ericsson sites to develop better software efficiently
- Background in telecommunication systems
- A standards-based communications-class server:
  - Open, standards-based common platform
  - High availability (greater than 99.999%)
  - Broad range of support for both infrastructure and value-added applications
  - Multimedia, network and application processing capabilities
  - Product life-cycle of 7 years

ABOUT ME

- Improving development tools with research projects, open source tools, tool vendors and other companies
- GDB improvements, non-stop, multi-process, global breakpoint, dynamic tracepoint, core awareness, OS awareness, … with CodeSourcery
- Eclipse GDB integration, debug analysis with CDT community e.g. WindRiver
- Linux tracing research project with Ecole Polytechnique (Prof. Michel Dagenais)

ABOUT ME

- Linux tracing: user space tracing, GDB integration, binary format, buffering scheme, … with EfficiOS (Mathieu Desnoyers)
- Eclipse Linux tracing integration and analysis with Red Hat
- Organizing Linux Tracing Summit:

2008: https://ltt.polymtl.ca/tracingwiki/index.php/TracingSummit2008

2009: http://www.linuxsymposium.org/2009/view_abstract.php?content_key=108

2010: http://events.linuxfoundation.org/events/linuxcon/minisummits

GDB - UBIQUITOUS DEBUGER

- Embedded system development usually requires different targets
- Switching to a different debugger each time is a mess
- One GDB/Eclipse binary on host can support
  - Multi architecture ->GDB target description <architecture> e.g. x86
  - Multi operating systems ->GDB target description <osabi> e.g. linux
  - Simulator/Emulator ->Same protocol, MI, Eclipse e.g. Simics
  - Unit test infrastructures -> Normal host base debug
  - UML models with code generation -> Normal host/target debug
  - Real target -> GDB stub (e.g. gdbserver on linux)
  - JTAG -> many JTAG devices work with GDB

MULTI-CORE-PROCESS-CONTEXT

- With multi-core more things are done in parallel in many processes
- Core awareness, i.e. which threads are running on which cores
- Application debug, attach to all processes of an application, step the application, step one core, etc.
- Follow child process created with a fork, exec, handles dynamic loading
- Many processes can potentially execute the same code, global breakpoint will attach to the process only when the breakpoint is hit

SPECIAL BREAKPOINT

- Conditional Breakpoint
  - Stop only if the condition is *true*.
  - C assert condition, i.e. breakpoint can happen when assertion is false
- Data Breakpoint or Watchpoint
  - Stop whenever the value of an expression change
  - Don't have to predict where this may happen
  - Can be a complex expression or just a single variable
- Program event breakpoint
  - Stop when a special event occurs
  - throwing or catching of a C++ exception, unhandled exception
  - call to exec, fork, close syscall

ALTERING EXECUTION

- A bug was found
- Test a correction without recompiling, e.g.:
  - store new values into variables or memory locations
  - send a signal
  - restart the program at a different address
  - call a function
  - code patching

OS AWARENESS

- Some programs have a deep interaction with the operating system

- Showing OS resources in the debugger can help e.g:

  process groups, processes, threads, file descriptors, internet-domain sockets, shared memory segments, semaphore, message queues, loaded kernel modules, etc.

- Not completed yet

NON-STOP

- Debugging a process by stopping its execution might cause the program to change its behavior drastically, or perhaps fail, even when the code itself is correct.
  - Troubleshooting in the lab
  - Chasing a race condition
  - Debugging problems happening only under heavy load
  - Investigating user interface issues

- Non-Stop allows to stop and examine one or more thread in the debugger <u>*while other threads continue to execute freely*</u>

DEBUG TRACEPOINT

- Sometimes it is not feasible to stop the execution of even one thread, e.g. live system
- Tracepoint collects user-specified info and continues execution *without stopping any thread*
- Dynamic i.e. inserted with a jump (in process), when a jump cannot be used, a trap between the process and the debug stub is used
- Data collection can be conditional to a user specified expression

DEBUG TRACEPOINT

- Tracepoint actions:
  - collect state trace data e.g. timestamp, and program data e.g. variables, register, memory
  - evaluate expressions , e.g. modify trace variables
  - step (similar to breakpoint step) and collect data in each step
- A trace experiment can be stopped after the n'th hit
- Static tracepoint data i.e. LTTng UST can also be stored in the debug tracepoint buffer
- Debug tracepoint are good when no static tracepoint are available and for small quantity of data
- A tracer (e.g. LTTng) should be used to collect GB of data

CHECKPOINT

- Save a snapshot of a program's state, including memory, registers, variables, etc.

- Can go back to the checkpoint, similar to a bookmark

- Cannot do things like step backwards

REVERSIBLE DEBUG

- Solving a bug is similar to solving a mystery, one needs to go back in time to understand what happened.
- When debugging, you realize that you have gone too far, and some event of interest has already happened.
- Undo the changes in machine state that have taken place as the program was executing normally i.e. variables, registers etc. revert to their previous values.

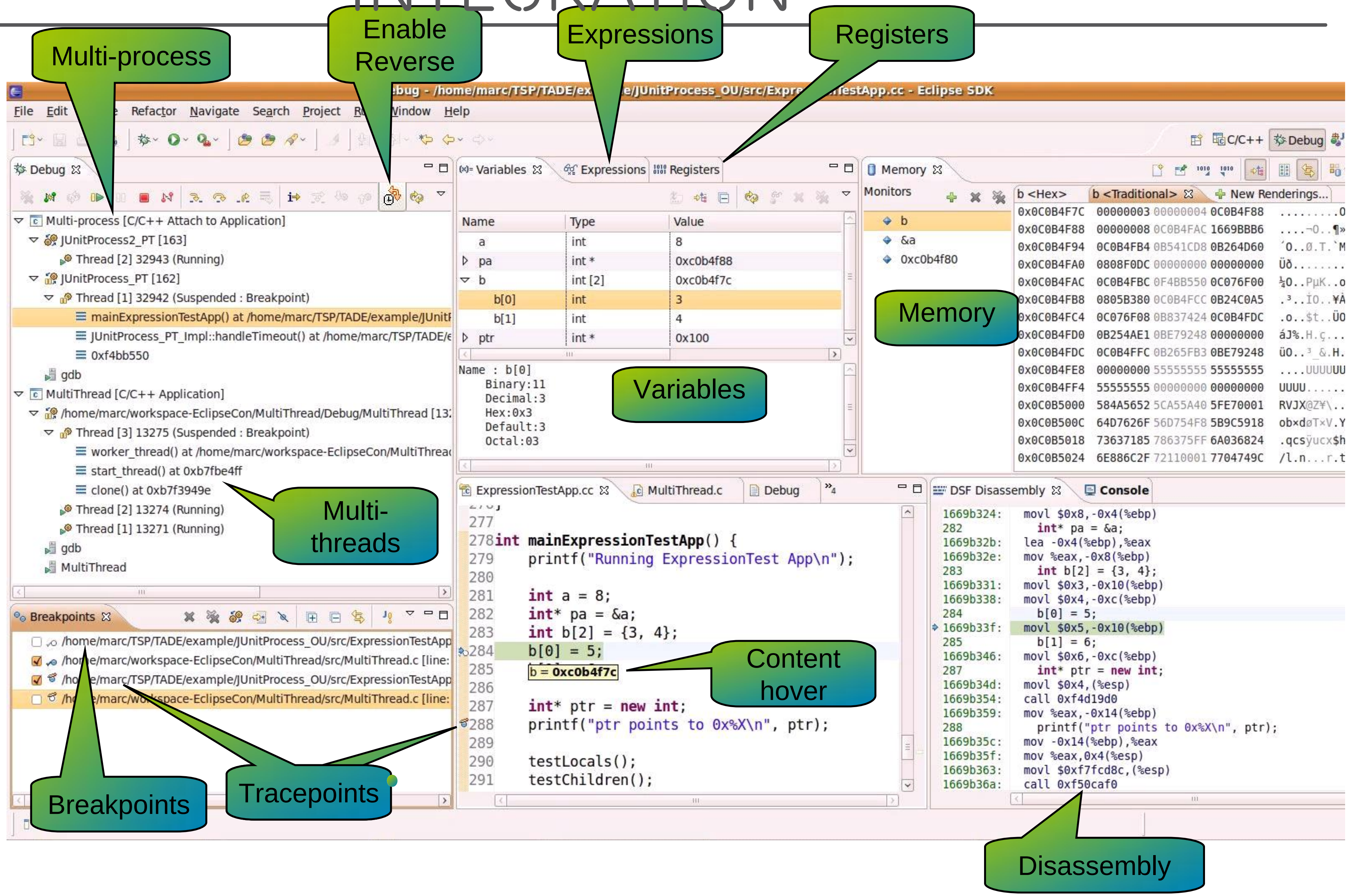
REVERSIBLE DEBUG

- Process record and replay on Linux
- Simulators are typically faster than process record/replay
- A simple example, a variable doesn't have the right value
  - add a watchpoint on the variable
  - set the debug in reverse
  - debugger will go back in time when the variable was last changed

DEBUGGER EXTENSIONS

- A new debug feature can be added quickly
- Two mechanisms for extensions
  - Command Files
  - Python scripting
- A complete new feature can be added via python scripting

ECLIPSE DEBUG INTEGRATION



ECLIPSE DEBUG INTEGRATION

EMBEDDED CHALLENGES

› Slow connection to target
  – Ethernet
  – JTAG
  – Serial Port

› More visibility into target hw
  – On-chip Peripherals
  – Processor Cache
  – Flash Memory
  – Tracing
  – Hardware Breakpoints

› Varied target hw architectures
  – Multiple Cores/CPUs/DSPs
  – Memory Models

ECLIPSE DEBUG FRAMEWORK

› Strict Concurrency Model
  • Complex caching techniques
  • Exact control over when and what data is retrieved from target
  • CMD Coalescing

› Modular Debugger Implementation
  • Selective re-use of a standard implementation
  • Custom services can be written to interact with custom hardware

› Decoupled view layout from data model
  • Views layout and content easily customized

MULTI-CONTEXT

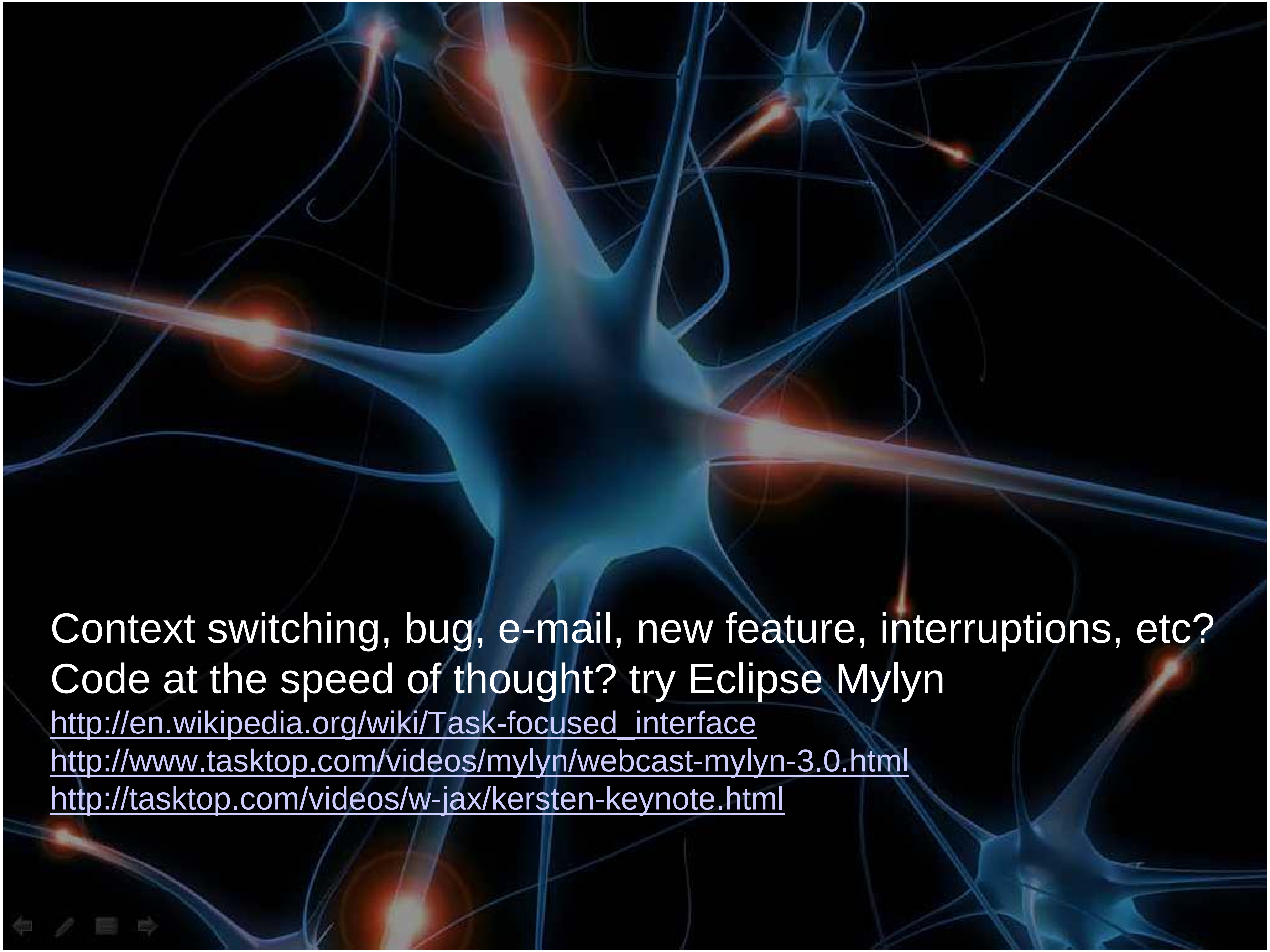
- Simultaneous debugging of multiple cores, processes, threads, any objects represented in the debugger views

- Improving the workflow of multi-context debugging, e.g. breadcrumb or one liner debug view, thousands of processes, etc.

- Come and join the fun
  http://wiki.eclipse.org/DSDP/DD/MultiContext

ECLIPSE IDE,
WHAT FOR?

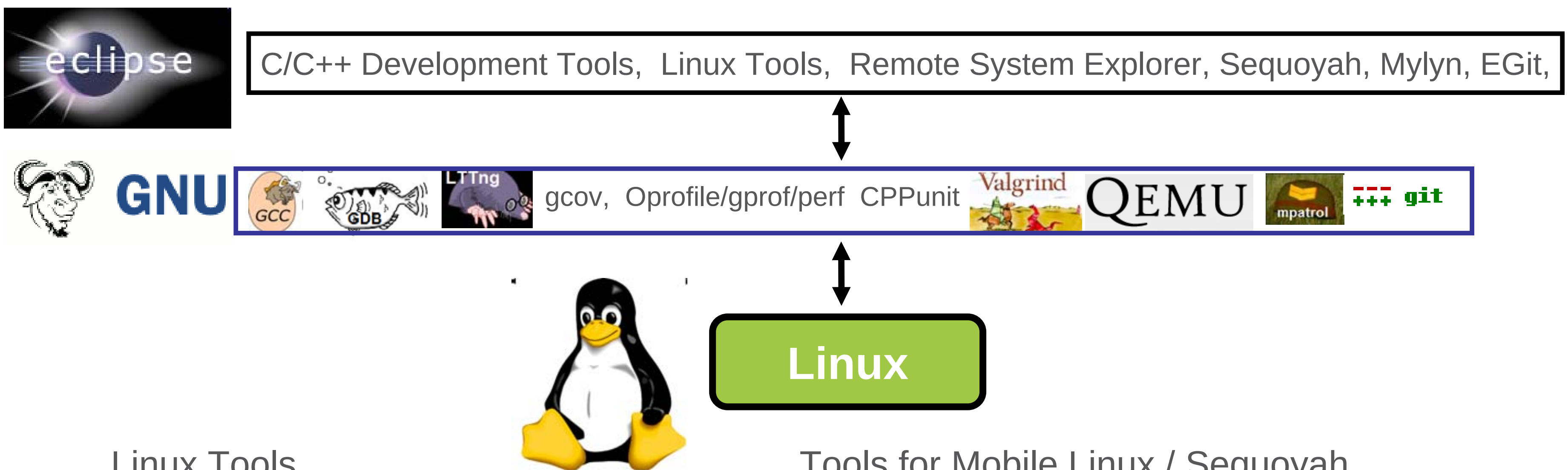
- Multi-core systems with multiple processes
- Debug multi-process, non-stop with cmd line?
- Performance analysis?
- What is your reason to use an IDE?



Context switching, bug, e-mail, new feature, interruptions, etc?
Code at the speed of thought? try Eclipse Mylyn
http://en.wikipedia.org/wiki/Task-focused_interface
http://www.tasktop.com/videos/mylyn/webcast-mylyn-3.0.html
http://tasktop.com/videos/w-jax/kersten-keynote.html

LINUX ECLIPSE PROJECTS

ERICSSON



Linux Tools
http://www.eclipse.org/linuxtools

C/C++ Development Tool
http://www.eclipse.org/cdt/

Target Management
http://www.eclipse.org/dsdp/tm

Parallel Tools Platform
http://www.eclipse.org/ptp/

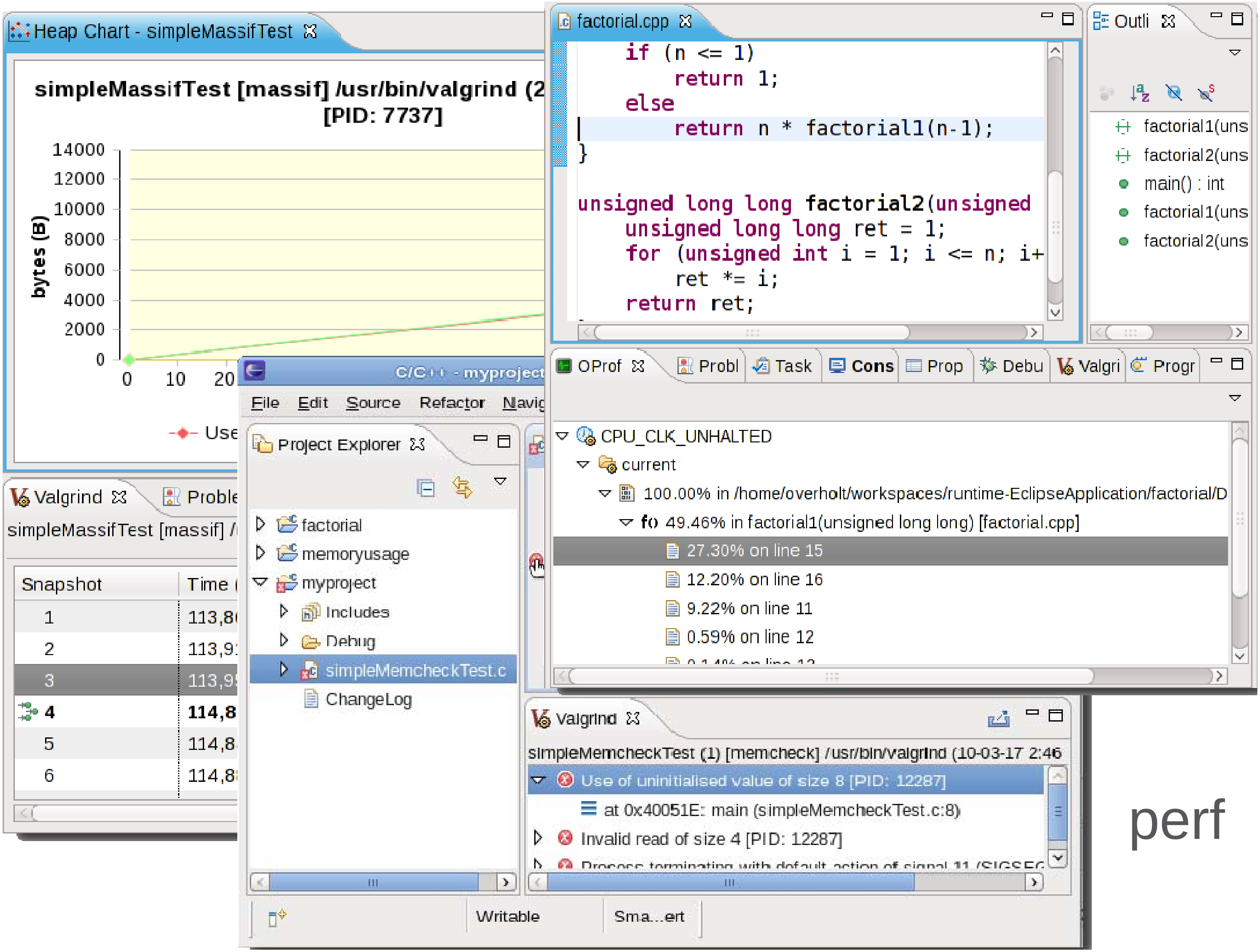
Tools for Mobile Linux / Sequoyah
http://www.eclipse.org/dsdp/tml

Mylyn, code at the speed of thought
http://www.eclipse.org/mylyn

EGit
http://www.eclipse.org/egit

Etc.
http://www.eclipse.org/projects/listofprojects.php

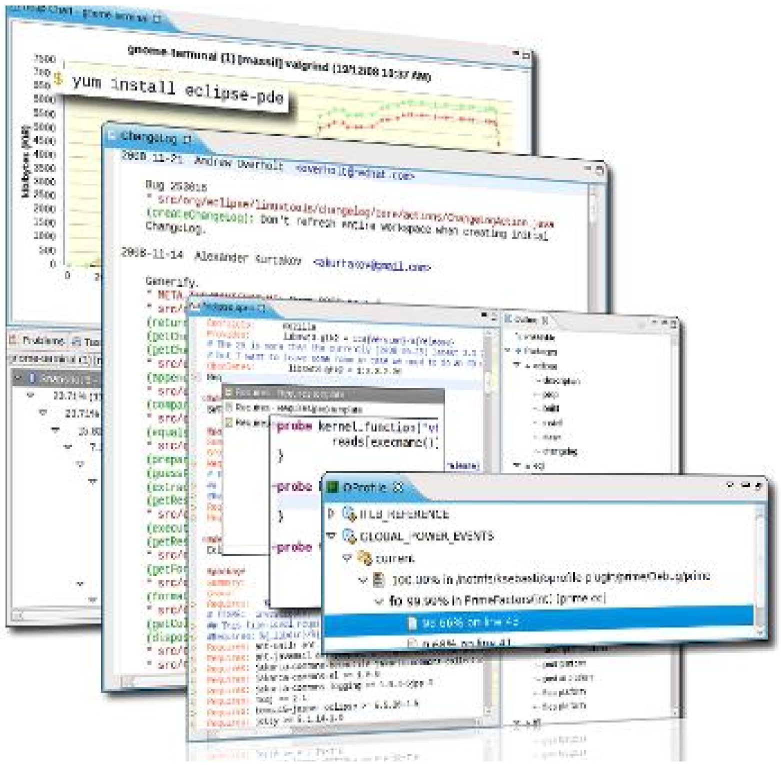
ECLIPSE FOUNDATION, 200 MEMBERS


Heap Chart - simpleMassifTest
simpleMassifTest [massif] /usr/bin/valgrind (2
[PID: 7737]
bytes (B)
14000
12000
10000
8000
6000
4000
2000
0
0
10
20
factorial.cpp
if (n <= 1)
return 1;
else
return n * factorial1(n-1);
}
unsigned long long factorial2(unsigned
unsigned long long ret = 1;
for (unsigned int i = 1; i <= n; i+
ret *= i;
return ret;
Outli
factorial1(uns
factorial2(uns
main() : int
factorial1(uns
factorial2(uns
C/C++ - myproject
File Edit Source Refactor Navig
Project Explorer
factorial
memoryusage
myproject
Includes
Debug
simpleMemcheckTest.c
ChangeLog
OProf Probl Task Cons Prop Debu Valgri Progr
CPU_CLK_UNHALTED
current
100.00% in /home/overholt/workspaces/runtime-EclipseApplication/factorial/D
49.46% in factorial1(unsigned long long) [factorial.cpp]
27.30% on line 15
12.20% on line 16
9.22% on line 11
0.59% on line 12
Valgrind
Proble
simpleMassifTest [massif] /
Snapshot
Time
1
113,8
2
113,9
3
113,9
4
114,8
5
114,8
6
114,8
Valgrind
simpleMemcheckTest (1) [memcheck] /usr/bin/valgrind (10-03-17 2:46
Use of uninitialised value of size 8 [PID: 12287]
at 0x40051E: main (simpleMemcheckTest.c:8)
Invalid read of size 4 [PID: 12287]
Writable
Sma...ert

perf

ECLIPSE LINUX TOOLS PROJECT

- Contribute/Get Involved
- Contact Us
- Downloads
- C/C++ Tools Projects
  - Autotools
  - Callgraph
  - ChangeLog
  - Libhover
  - Man Page
  - LTTng
  - OProfile
  - Systemtap
  - Valgrind
- Packaging/Distribution Projects
  - eclipse-build
  - RPM Stubby
  - Specfile Editor
  - Distribution Packaging Status



The Linux Tools project aims to bring a **full-featured C and C++ IDE** to Linux developers. We build on the source editing and debugging features of the CDT and integrate popular native development tools such as the GNU Autotools, Valgrind, OProfile, RPM, SystemTap, GCov, GProf, LTTng, etc. Current projects include Autotools build integration, a Valgrind heap usage analysis tool, and an OProfile call profiling tool. We also have projects implementing LTTng trace viewers and analyzers.

The project also provides a place for Linux distributions to collaboratively overcome issues surrounding distribution packaging of Eclipse technology. The project produces both best practices and tools related to packaging. Since our 0.3.0 release, one of our features is a source archive of the Eclipse SDK that can be used by all Linux distributions building and distributing it.

Downloads

Get our latest **0.5** release (*2010-03-18*)!

Get Involved

Find out how you can get involved with the project

- Managed build for various toolchains, standard make build
- Source navigation, type hierarchy, call graph, include browser, macro definition browser, code editor with syntax highlighting, folding and hyperlink navigation,
- Source code refactoring, static analysis
- Visual debugging tools, including memory, registers, and disassembly viewers

TRACING (LTTNG)

Static Tracepoint:

- Created by designer *before compilation* at *development time*
- Static tracepoints represent *wisdom of developers* who are most familiar with the code
- Helps developers to think about tracing (using only trial-error dynamic traces is not efficient)
- The rest of the world can use them to extract a great deal of useful information *without having to know the code*

TRACEABLE DATA

- Everything should be traceable
- User space
- Kernel
- Non-Maskable Interrupt (NMI)
- Thread and signal safe
- Events may not be lost
- Collect large trace data > 10GB

LOW OVERHEAD

- Low overhead is key, better tracing means more troubleshooting in field and quicker resolution of problems
- Very efficient probes with static jump, no trap, no system call
- Almost <u>*zero performance impact with instrumentation points disabled*</u> (kernel: static jump, userspace: uses fast boolean evaluation)
- <u>*Enable instrumentation points have low performance*</u> impact, i.e. a fast C function call
- Zero copy from event generation to disk write

TIME

- Accurate event ordering is key to enable trace synchronization or correlation of traces from
  - different CPU, cores
  - traffic exchanged between nodes
  - virtual machine, etc.
- LTTng timestamp precision is typically ~1ns i.e. cycle counter

TRACE DATA STORAGE

- Trace data is initially stored in shared memory buffers
- Tracing daemon then writes to the chosen trace-store:
  - circular “flight recorder” buffer
  - local disk
  - remote disk
  - remote stream, e.g. live monitoring
- Binary trace format highly optimized for compactness
- Self describing trace format
- Generate events with arbitrary number of arguments, variable event sizes

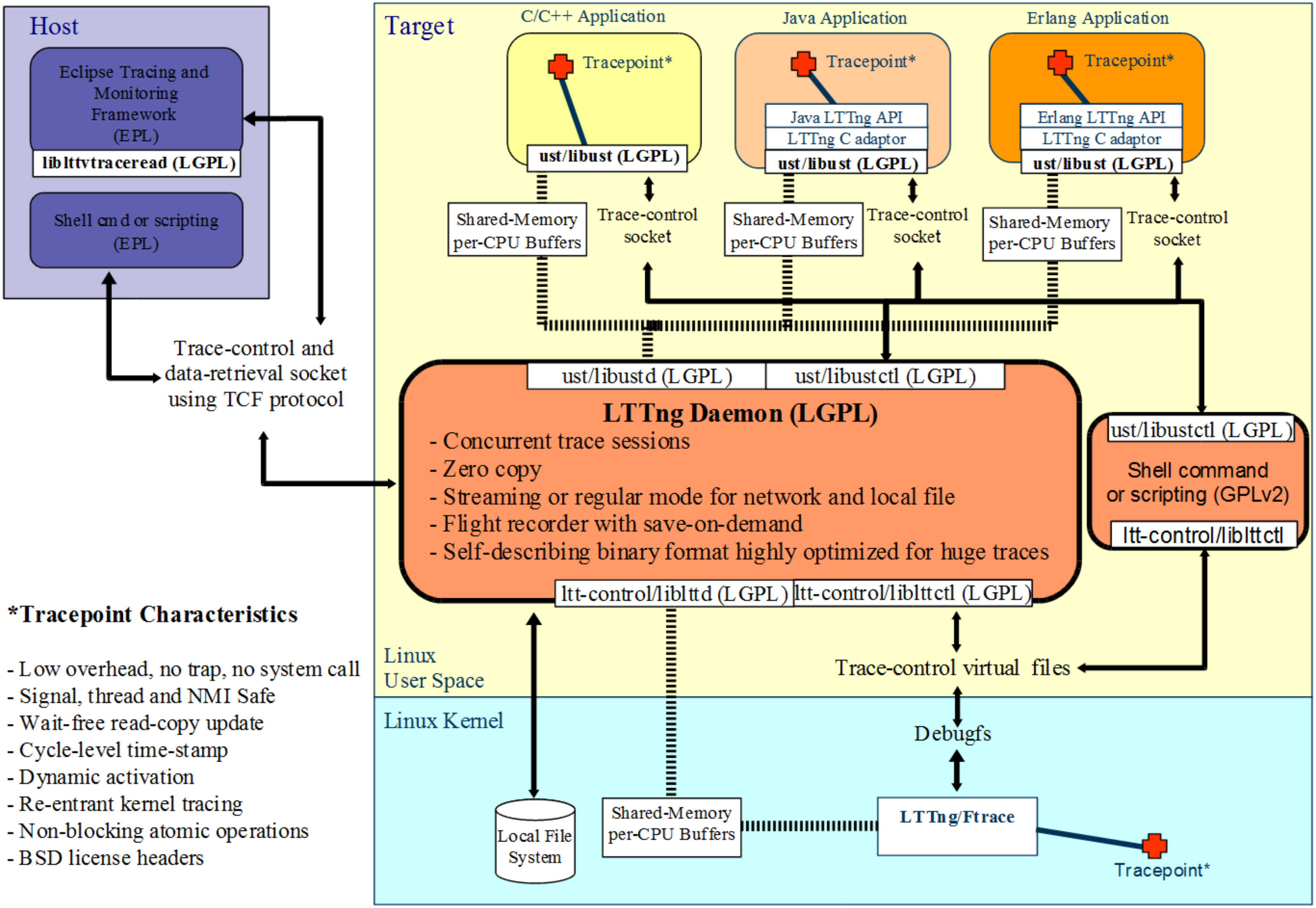
SCALABILITY

- Scalable to high core numbers
- Wait free Read-Copy-Update mechanism
- Per-CPU buffers
- Non-blocking atomic operations
- Simultaneous recording of multiple traces
  - system administrator monitoring
  - field engineered to troubleshoot a specific problem
- Performance is more than 5 times better than dynamic tracing (e.g. with trap), this margin is increasing on systems with more cores

USER SPACE TRACING

- Very low disturbance, highly scalable
- Uses user-space Read-Copy Updates (RCU) wait-free synchronization to trace events without requiring any system call or trap, i.e. *same proven algorithms as kernel tracer*
- User space independent from the kernel tracer to ease integration, distribution, port
- Conditional tracing in userspace

LTTng Low-Overhead Tracing Architecture

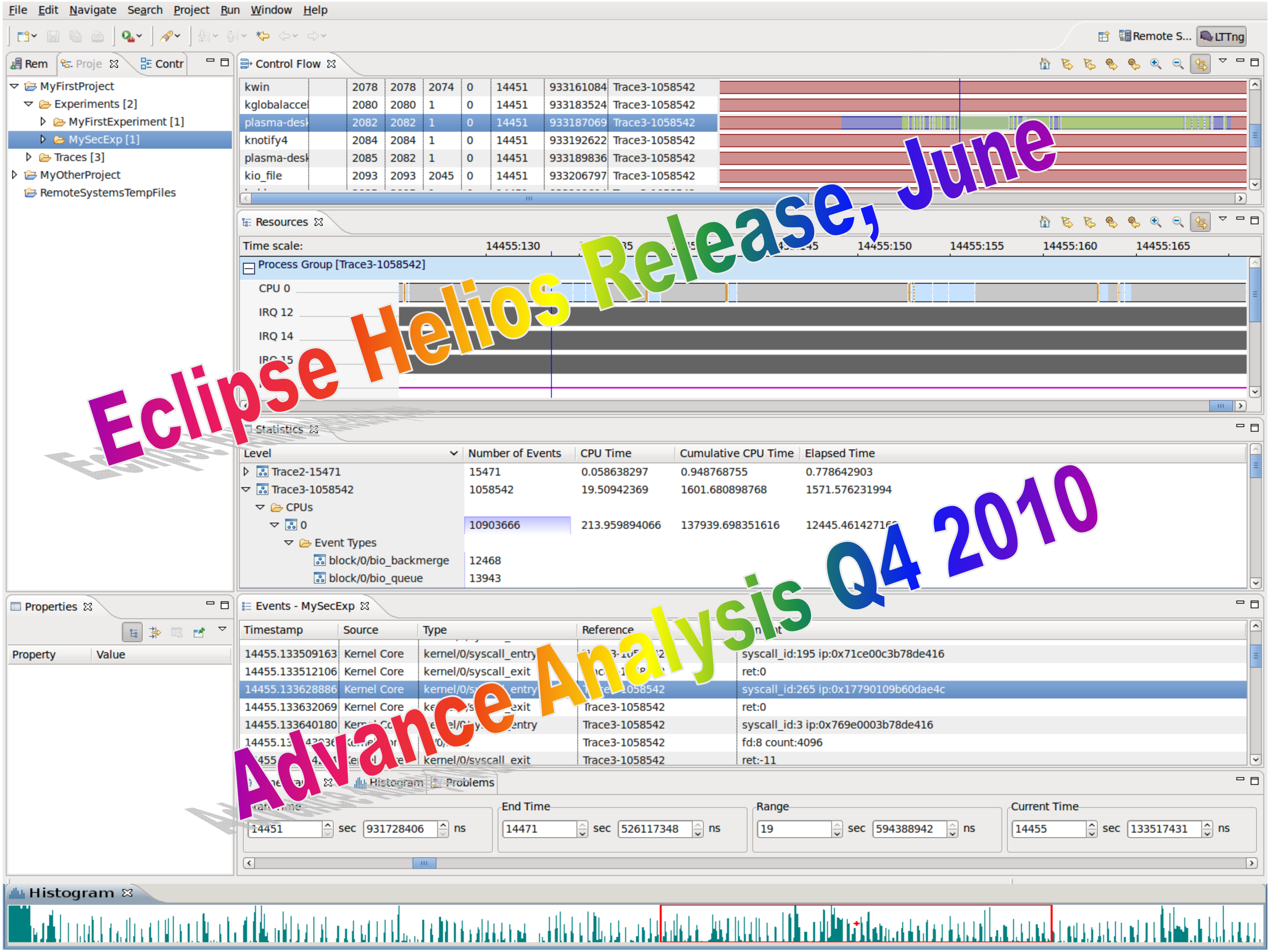


***Tracepoint Characteristics**

- Low overhead, no trap, no system call
- Signal, thread and NMI Safe
- Wait-free read-copy update
- Cycle-level time-stamp
- Dynamic activation
- Re-entrant kernel tracing
- Non-blocking atomic operations
- BSD license headers

ANALYSIS

- Resource view
- Per thread execution state (control flow view)
- Event rate histogram
- Detailed event list, filtering
- View synchronization
- IRQ latency



Eclipse Helios Release, June
Advance Analysis Q4 2010
File Edit Navigate Search Project Run Window Help
Remote S...
LTTng
Control Flow
Resources
Statistics
Properties
Events - MySecExp
Histogram
Problems
MyFirstProject
Experiments [2]
MyFirstExperiment [1]
MySecExp [1]
Traces [3]
MyOtherProject
RemoteSystemsTempFiles
Time scale:
Process Group [Trace3-1058542]
CPU 0
IRQ 12
IRQ 14
Level
Number of Events
CPU Time
Cumulative CPU Time
Elapsed Time
Trace2-15471
Trace3-1058542
CPUs
Event Types
block/0/bio_backmerge
block/0/bio_queue
Timestamp
Source
Type
Reference
Property
Value
End Time
Range
Current Time

ANALYSIS

- Trace synchronization
  - Time correction
  - Multi-core
  - Multi-level
  - Multi-node, distributed
- Dependency analysis, delay analyzer
  - Dependencies among processes
  - How total elapsed time is divided into main components
- Pattern matching
  - Security
  - Performance
  - Testing lock acquisitions
- Correlation
  - Other format
  - Text base logs
  - Multi-level

MULTI-CORE TROUBLESHOOTING

- Major software redesign is normally required to benefit from multi-core architectures

- Software development industry and individual developers are facing problems whose resolution requires to understand the interaction between all layers, including third party products e.g.
  - Hypervisor
  - Operating system
  - Virtual machines
  - System libraries
  - Applications
  - Operation and maintenance
  - Many Languages: C/C++, Java, Erlang

COMPLEX SYSTEMS

- A typical system these days:
  - Linux on a few cores
  - Low-level RTOS on another core
  - DSP's, etc.

- Developed in different context,
  - In-house development
  - Consultant
  - Reusable components
  - Third party products

- Understanding what is happening on the system requires compatible tools, i.e. de facto standard

LINUX TOOL WORK GROUP?

- Open source contributions are growing exponentially, contributions are sometimes incompatible or result in duplicated work:
  - Many forks of GDB
  - competing projects have emerged, e.g. frysk, EDC
  - Linux trace initiatives e.g. LTTng, ftrace, perf, utrace, SystemTap
  - Very hard to plan cross project features

- Let's take this to the next level
  - not only contribute the parts needed for one company, plan together
  - avoid incompatible data, inconsistent work, and duplicated efforts
  - e.g. Executable and Linkable Format (ELF), DWARF debug format
  - create an industry de-facto standard for tools, reference implementation
  - Show it’s easy to add features to tools
  - Budget cycle! Ecosystem of tool improvements, support
  - Linux foundation tool work group?

WE CAN DO BETTER THAN PRINTF

ERICSSON